Vibe-JIT-ITP

Mirek Olšák

How I got here

- 2023: Pylogic – a python logic experiment
  - Not caring about minimal foundations
  - Trusting ATP’s
- Dilemma: Speed / Convenience / Trust
  - Julia: We are greedy, we want more

Dream

- Minimalistic kernel
- Fast & maximally bootstrappable
- Prove (almost) its own correctness

Julia: JIT

- Just
- In
- Time
  compilation

Dream: JIT

- Just
- In
- Trust

  computation

As soon as we can prove a piece of byte code is safe,
kernel is willing to trust it

Disclaimer:

I started two weeks ago,
do not expect too much yet ;-)

Method

Two ways of using AI assistants

- Code I care about
- Code where I only care about the outcome

Code I care about (~3k LOC)

```
File  Edit  Options  Buffers  Tools  C  Help
  return thm;
}

struct theorem *logic_add_axiom(struct term *statement) {
  return _theorem_new(logic_term_retain(statement));
}
void logic_theorem_free(struct theorem *thm) {
  if (!thm) return;
  logic_term_free(thm->p);
  free(thm);
}

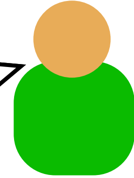
/*
  The backbone logic.
 */

struct theorem *logic_modus_ponens(struct theorem *thm_impl, struct theorem *thm_prem
ise) {
  if (!thm_impl || !thm_premise) return NULL;
  if (thm_impl->p->symbol != &logic_impl_symbol) return NULL;
  if (!logic_term_eq(thm_impl->p->data.args[0], thm_premise->p)) return NULL;
  return _theorem_new(logic_term_retain(thm_impl->p->data.args[1]));
}

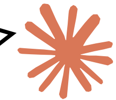
struct theorem *logic_thm_instantiate(struct theorem *thm, struct symbol *fvar, struc
t term *value) {
  if (!thm || !fvar || !value) return NULL;
  if (fvar->kind != FVAR || value->bvar_depth > fvar->arity) return NULL;
  return _theorem_new(logic_term_substitute_fvar(fvar, value, thm->p));
}

/*
  Literal manipulation.
 */
-:---  logic.c        61%   L533  Git:master  (C/*l +2 Abbrev)
```

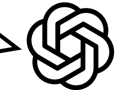
Review my code



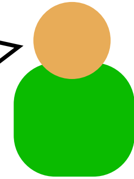
Bug 1, Bug 2, ...



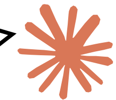
Bug 1, Bug 2, ...



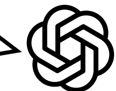
I addressed the issues



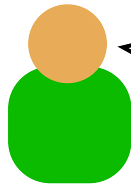
Now OK



Bug 2 remains ...



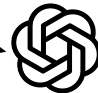
Care only about outcome (~90k LOC)



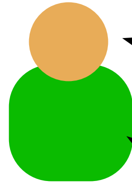
Prove this

```
challenge setRec_expand(a, b prev : rec(b, prev)) :
  setRec(a, b prev : rec(b, prev)) =
  rec(a, funBuild(a, x : setRec(x, b prev : rec(b, prev))))

challenge set_induction(a, x : p(x)) :
  all(x : all(y : y IN x -> p(y)) -> p(x)) -> p(a)
```



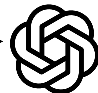
I need more library lemmas



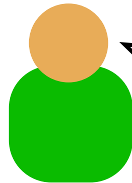
Prove them

I need better automation

Ok, implement it



I made some progress...



Good, continue. Only stop at a kernel issue.

Kernel Logic

- Base rules:
  - Modus Ponens
  - free variable instantiation
- No sorts, “true” / “false” are two objects
- No native context handling:

```
axiom impl_weaken(a, b) : a -> (b -> a)
axiom impl_chain(a, b, c) : (a -> b -> c) -> (a -> b) -> a -> c
```

Kernel Logic

- Second order free variables

```
axiom eq_rfl(a) : a = a
axiom eq_congr(a, b, x : p(x)) : a = b -> p(a) -> p(b)

axiom ex_intro(a, x : p(x)) : p(a) -> ex(x : p(x))
axiom ex_elim(x : p(x)) : ex(x : p(x)) -> p(choose(x : p(x)))

def all(x : p(x)) = not(ex(x : not(p(x))))
```

- Axioms of set theory with 999999999999 universes
- Few literal manipulations: +, *, /, len, slice

JIT

```
declare isSafeCode(code, input1, input2, outLen)
declare executedTo(code, input1, input2, output)
```

- Design choices
  - Non-determinism ok
  - Infinite loop ok
  - Stack overflow ok
  - Any other crash / undefined behavior: wrong

x86_64 spec

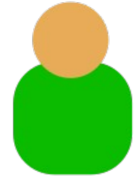
```
File  Edit  Options  Buffers  Tools  C++  Help
        auto rm = decode_rm(pc + 1, modrm, rex, /*is_byte=*/true);
        uint8_t byte_val = regs[src] & 0xff;
        std::cout << "[DEBUG] mov " << rm_name(rm) << ", " << RegNames[src]
                  << "l (= 0x" << std::hex << (int)byte_val << ")" << std::endl;
        rm_write_byte(rm, byte_val);
        regs[RIP] = pc + 2 + rm.extra;

      } else if (b == 0x89) { // mov r/m, r  (32-bit or 64-bit)
        uint8_t modrm = mem_at(pc + 1, Access::Exec);
        int src = get_reg_idx((modrm >> 3) & 7, rex.r);
        auto rm = decode_rm(pc + 1, modrm, rex);
        uint64_t val = rex.w ? regs[src] : regs[src] & 0xffffffff;
        std::cout << "[DEBUG] mov " << rm_name(rm) << ", " << RegNames[src]
                  << " (= 0x" << std::hex << val << ")" << std::endl;
        rm_write(rm, val, rex.w);
        regs[RIP] = pc + 2 + rm.extra;

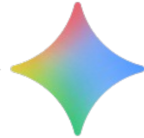
      } else if (b == 0x8a) { // mov r8, r/m8
        uint8_t modrm = mem_at(pc + 1, Access::Exec);
        int dst_field = (modrm >> 3) & 7;
        check_no_high_byte(dst_field, rex);
        int dst = get_reg_idx(dst_field, rex.r);
        auto rm = decode_rm(pc + 1, modrm, rex, /*is_byte=*/true);
        uint8_t byte_val = rm_read_byte(rm);
        std::cout << "[DEBUG] mov " << RegNames[dst] << ", " << rm_name(rm)
                  << "l (= 0x" << std::hex << (int)byte_val << ")" << std::endl;
        regs[dst] = (regs[dst] & ~(uint64_t)0xff) | byte_val; // preserve bits 8-63
        regs[RIP] = pc + 2 + rm.extra;

      } else if (b == 0x8b) { // mov r, r/m  (32-bit or 64-bit)
        uint8_t modrm = mem_at(pc + 1, Access::Exec);
        int dst = get_reg_idx((modrm >> 3) & 7, rex.r);
        auto rm = decode_rm(pc + 1, modrm, rex);
U:---  emulator.cpp   37%   L419  Git-master  (C++//l +2 Abbrev)
```

Write a bytecode and an emulator



Here you go!



What is the difference between 89 / 8a?

...

Axioms vs. Defs

```
def A = ...
def B = ...
def C = ...
def X = ...
...
axiom safe_spec : X(code, input1, input2, outLen) ->
  safeCode(code, input1, input2, outLen)
```

```
declare A = ...
axiom A_spec : ... A ...
declare B = ...
axiom B_spec : ... B ...
def C = ...
def X = ...
...
axiom safe_spec : X(code, input1, input2, outLen) ->
  safeCode(code, input1, input2, outLen)
```

Selected proven facts

- Recursion on sets / natural numbers

Selected proven facts

- Recursion on sets / natural numbers
- Integers form an ordered ring

Selected proven facts

- Recursion on sets / natural numbers
- Integers form an ordered ring
- Tuple concatenation form a monoid

Selected proven facts

- Recursion on sets / natural numbers
- Integers form an ordered ring
- Tuple concatenation form a monoid
- Binary XOR exists on natural numbers

Selected proven facts

- Recursion on sets / natural numbers
- Integers form an ordered ring
- Tuple concatenation form a monoid
- Binary XOR exists on natural numbers
- Safe to run: 49 c7 00 de ad be ef c3

Proof-Language Automation

User-facing structure

- Goal-aware proof terms: `show`, `apply`, `claim`, `calc`.
- Local contexts, existential witnesses, cases, and binder-aware blocks.
- Continuation syntax keeps nested binder arguments readable.

Reusable proof engines

- `auto_fact`: theorem-driven closure facts and type-like facts.
- `apply ... using`: goal-directed theorem instantiation.
- Binder-aware rewriting via extensionality lemmas.

```
theorem refl_example : 42 = 42 :=
  eq_rfl(42);

theorem mp_example(a, b) :
  (a -> b) -> a -> b :=
{
  assume hab : a -> b;
  assume ha : a;
  hab @ ha
}

# goal: A = B
apply set_ext_intro_elim using (
  fix(x : split(auto, auto)))

claim hp : all(x : p(x)) :=
  fix(x : show(p(x), auto));

obtain(hex, witness hw :
  show(q(witness), auto))
```

Automation searches and organizes, but still reconstructs kernel-checkable proof terms.

Mathematical Automation

Equality and normal forms

- Conservative definitional equality with bounded unfolding.
- Rewriting under binders using synthesized extensionality theorems.
- AC normalization for registered operators.
- Tuple concatenation normalized as a proved monoid.

Arithmetic and sets

- Guarded integer simplification, literal evaluation, order facts.
- Divisibility closure and modular-arithmetic lemmas.
- Extensionality and transitive-closure induction.

```
show((a + b) + c = (a + c) + b,
  arith_ac_norm_eq((a + b) + c, (a + c) + b))

# context: a < b, b <= c
claim hlt : a < c := order_fact(a < c);

claim hdiv : divisible(a * b, a) :=
  divisibility_fact(divisible(a * b, a));

show(f ++ zero = f,
  tuple_norm_eq(f ++ zero, f))

claim hmono : a + b = b + a :=
  arith_simp_eq(a + b, b + a);
```

Concrete Data and Symbolic Execution

Binary data

- Concrete literal facts: length, slices, bytes, numeric values.
- Byte-function extensionality for `litBytes`, `natBytes`, `tup1`, and `++`.
- Prefix/segment mapping helpers for executable memory.

x86_64 instruction skeletons

- Build `runInstructionRex` from `nextWorldState`.
- Project safety/results for deterministic, flag-changing, and push instructions.
- Used by the JIT demo and larger bytecode proofs.

```
theorem byte_segment_ok(state) :
  iniWorldState(```00 00 00 00 00 00 00 c3```,
    in1, in2, 8, state) ->
  memoryMapped(state, state @ "rip" + 7,
    litBytes(```c3```)) :=
{
  assume hini : iniWorldState(
    ```00 00 00 00 00 00 00 c3```,
    in1, in2, 8, state);
  show(memoryMapped(state, state @ "rip" + 7,
      litBytes(```c3```)),
    ini_code_segment_mapped(7, hini))
}

show(safeWorldState(state2),
  instruction_safe(hdet, hsafe,
    hnext, hmap, hexec,
    tuple_bytes, nonempty, not_rex))
```

Takeaway: one automation layer spans abstract mathematics, concrete bytes, and machine-code symbolic execution.
```

How far we got (so far)

How far we got (so far)

- 4 244 061 667 345 952 017 is a prime number

How far we got

- 4 244 061 667 345 952 017 is a prime number

````
def primeTestCode = ```
  # prologue
  53                # push rbx
  48 8b 1f          # mov rbx, [rdi]
  b9 07 00 00 00    # mov ecx, 7

  # loop1 (offset 9)
  48 8b c3          # mov rax, rbx
  31 d2             # xor edx, edx
  48 f7 f1          # div rcx
  48 85 d2          # test rdx, rdx
  74 2e             # je not_prime  (+46)
  48 3b c1          # cmp rax, rcx
  76 06             # jbe loop2_start (+6)
  48 83 c1 06       # add rcx, 6
  eb e8             # jmp loop1  (-24)

  # loop2_start (offset 33)
  b9 05 00 00 00    # mov ecx, 5

  # loop2 (offset 38)
  48 8b c3          # mov rax, rbx
  31 d2             # xor edx, edx
  48 f7 f1          # div rcx
  48 85 d2          # test rdx, rdx
  74 11             # je not_prime  (+17)
  48 3b c1          # cmp rax, rcx
  76 06             # jbe prime  (+6)
  48 83 c1 06       # add rcx, 6
  eb e8             # jmp loop2  (-24)

  # prime (offset 62)
  41 c6 00 01       # mov byte [r8], 1
  5b                # pop rbx
  c3                # ret

  # not_prime (offset 68)
  41 c6 00 00       # mov byte [r8], 0
  5b                # pop rbx
  c3                # ret
```
````
````

Thank you for your attention

https://git.olsak.net/mirek/Vibe-ITP